Accelerating Poly1305 Cryptographic Message Authentication on the z14

Umme Salma Gadriwala
McMaster University
gadriwau@mcmaster.ca

Christopher Kumar Anand
McMaster University
anandc@mcmaster.ca

Curtis D'Alves
McMaster University
dalvescb@mcmaster.ca

Bill O'Farrell
Linux on IBM Z Open Source Ecosystem
IBM Canada Lab, Markham
billo@ca.ibm.com

ABSTRACT

In this paper, we examine the implementation and acceleration of the Poly1305 authentication algorithm on the recently announced IBM z14 computer. Two approaches are undertaken to improve performance of this important cryptographic algorithm. First, we restructure the algorithm to take advantage of a new instruction, VMSL, which employs floating-point hardware to perform high-speed high-throughput multiplications on integer limbs (big-integer digits) of large integers. With VMSL, we are able to eliminate multiplication as the dominant operation in Poly1305. Second, we apply Coconut, an extensible domain-specific language (DSL) embedded in Haskell, to generate a better schedule for parts of the algorithm that are performance bottlenecks. This combined approach has implications beyond Poly1305, as the same techniques can be applied to other cryptographic algorithms, such as elliptic curve digital signature algorithm (ECDSA) used in HyperLedger Blockchain.

1. Poly1305

Poly1305/ChaCha20 is an encryption cypher system in the class AEAD (Authenticated Encryption with Additional Data). AEADs support two operations: "seal" and "open". The seal operation encrypts the message using a secret key, and signs it with a Message Authentication Code (MAC). The open operation unencrypts the data and verifies that the MAC is valid for the message and the key. ChaCha20 is a stream cipher that fulfils the encryption function of the AEAD, while Poly1305 is a signing algorithm for generating the MAC. These two algorithms are nearly always paired together, and are usually considered as a unit. Poly1305/ChaCha20 is one of the two common AEADs, the other being AES-GCM. Poly1305/ChaCha20 is especially popular for use on mobile devices as it can be efficiently calculated with modest hardware resources. For a platform like the z14, Poly1305/ChaCha20 is of interest primarily because Z is designed for high-throughput, simultaneous I/O connections by thousands of clients, especially mobile clients. This paper is primarily concerned with Poly1305 (as opposed to ChaCha20) as the authors have been investigating a new approach to computing the Poly1305 MAC on z14.

Poly1305 creates a MAC by computing a univariate polynomial over a prime field. Specifically, the equation computed, as described by Bernstein in [1], is

$$(((m[0]r^{l} + m[1]r^{l-1} + \cdots + m[l-1]r) \bmod (2^{130}-5)) + s) \bmod 2^{128}$$

where m[0]…m[l -1] are modified 16 byte message blocks. Block m[l -1] can be shorter, but by convention is a whole number of bytes. The modification applied to message blocks is to have 2n added to them, where n is the block length in bits. This is done to prevent cryptographic attacks based on the possible presence of leading zeros.

In the equation, r is the first 16 bytes of a 32-byte key, and s is the second 16 bytes of that key. As formally expressed in this algorithm, all quantities are in little-endian order, and need to be converted to big-endian on machines, such as Z, which perform operations on big-endian quantities. The MAC produced (which is 16 bytes in length) must also be returned in little-endian order.

It should be noted that all cryptographic algorithms are potentially subject to "side-channel" attacks, where information is extracted from externally visible but unintentional behaviors of the algorithm. Although there are certain exotic approaches, such as

CASCON, Markham ,Ontario, Canada

ACM Reference format:
Gadriwala, U., Anand, C., D'Alves, C., and O'Farrell, B., In *Proceedings of CASCON 2017, Markham ON, Nov. 2017*, 8 pages.

measurements of heat dissipation in processor chips, side-channel attacks usually take the form of observations of nuances in timing – the length of time consumed for messages with various properties. In order to circumvent these attacks, the time for the algorithm as a whole can only vary with respect to message length, never with message contents. Thus, the algorithm should contain nothing of the form "if x then…" where x is anything other than message length (or number of bytes/blocks remaining in the message). Furthermore, all operations performed in a cryptographic algorithm must be individually constant time. As outlined below, this is an important consideration in the approach to computing Poly1305.

All of the work in this paper has been conducted with respect to cryptography in the Go programming language, although the results are generally applicable. All algorithms have been implemented in Go assembly language, with Go language interfaces. The reason for selecting Go for this work is that Go is the implementation language for HyperLedger Blockchain, the foundation for IBM High Security Business Network (HSBN). It is intended that some results of this research will be applied to higher-performing cryptographic algorithms for HSBN, including various elliptic curve computations (not just Poly1305).

2. Limbs and Prime Fields

For cryptographic purposes, a prime field is a finite field comprised of whole non-negative integers which are remainders of a prime number. The specific prime number utilized for Poly1305 is $2^{130} - 5$, which has, among other things, the desirable property that it is numerically close to a power of 2. It is also large relative to the size of a standard computer word, which is desirable cryptographically, but presents challenges in the computation of the Poly1305 equation. Calculating the MAC requires numerous multiplications, but the fact that the arithmetic is over a prime field means that the multiplications can be staged iteratively. The simplest (not the fastest) approach is to re-write the core multiplications of the Poly1305 equation as:

$$\Big(\big((m[0]r) + m[1]\big)r + m[2]\Big)r \ldots + m[l-1]\Big)r$$

This form of the equation can be staged as iterations on an accumulator which is initialized with m[0]. Each iteration involves a multiplication by r, but because the entire equation is to be mod($2^{130} - 5$), the mod operation can be distributed throughout the equation, and applied on every iteration. This step is called reduction. If we designate reduction as $\varrho[\ldots]$, then the equation becomes:

$$\varrho[(\varrho[(\varrho[(\varrho[(m[0]r)] + m[1])r] + m[2])r] \ldots + m[l-1])r]$$

Each iteration therefore has a multiplication and a reduction. We will outline below various re-formulations of the equation which lend themselves better to high-performance computation, however we will first look at basics.

In order to make these operations workable and fast on modern computers there are two issues that must be dealt with:

- The large size of the numbers – the inputs *m[n]* and *r* can be considered 130 bit numbers
- Computing the reduction operation through a division and remainder operation is far too expensive to be practical on most computers.

The solutions to both of these issues are related: express the large binary numbers as digits in a very-large-radix number system. In this context, the digits are called "limbs." For example, a common choice for the size of limbs for Poly1305 is 26 bits; in other words, the radix of the number system used is 2^{26}.With this size, five limbs are required to cover 130 bit numbers. Multiplication of whole 130 bit numbers can be broken-down into groups of multiplications with products that fit within 64 bits. Reduction consists of taking the overflow bits from each limb and adding them to the next higher-value limb. For overflow bits in the highest-value limb, they can be added to the lowest value limb using a trick that results from the fact that we are using modular arithmetic. Consider the number x, which represents the overflow bits in the highest value limb.

$$(x * 2^{130}) \bmod (2^{130} - 5) = \big(x \bmod (2^{130} - 5)\big) * 5 = x * 5$$

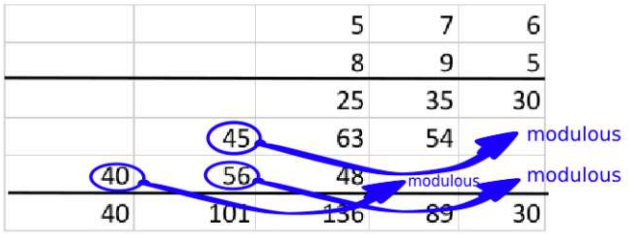
Thus, by multiplying those bits by 5, we can then add them to the lowest valued limb. The resulting overflow of the lowest valued limb can be added to the next limb and so on. A small fixed number of iterations is enough to fully, or nearly fully reduce the result. The reduction can even be carried out with non-adjacent limbs to mitigate register dependencies. For example, Bernstein suggests, for limbs h_0 through h_4, the sequence:
$h_0 \rightarrow h_1$, $h_3 \rightarrow h_4$, $h_1 \rightarrow h_2$, $h_4 \rightarrow h_0$, $h_2 \rightarrow h_3$, $h_0 \rightarrow h_1$, $h_3 \rightarrow h_4$.

3. Comba Multiplication with Modulus

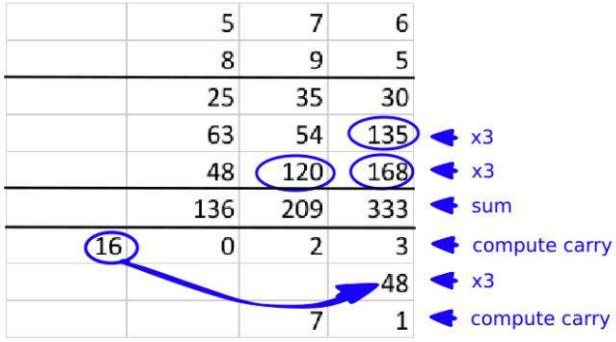
Once the message blocks and key multiplicands are broken into limbs, it is necessary to work out how to do the multiplications. The method usually employed is "Comba" multiplication [9] with modulus – which is just old-fashioned "schoolbook" multiplication modified in two ways that is also explained in [2]. First, carries are done lazily at the end of the multiplication, and second, overflows are handled with the same trick we showed in the previous section. For ease of explanation, consider an example in radix 10 using a prime of 997 (the "remainder trick" value is 1000-997 = 3). Let us imagine that we wish to multiply 576 * 895. Note that

(576 * 895) mod 997 = 71

The structure of the multiplication would be as thus:



Carrying out the modulus operands and adding, then applying modulus again, would yield the following:



giving the final answer of 71.

One thing that is apparent in this example is that the number of individual multiplications is the square of the number of digits. For Poly1305, that translates to the square of the number of limbs. If there are five limbs, there will be 25 multiplications required, plus all the associated additions and reductions. Bernstein and Schwabe have suggested a vectorization technique in [13] (which will be discussed in more detail below) which can use SIMD-parallel operations to reduce the number of multiplications to an average of 12.5 per 16-byte message block. However, one effective way to reduce multiplications is to reduce the number of limbs. For instance, reducing the number of limbs to three from five will reduce the raw number of multiplications from 25 to nine. We will discuss below how this also can be vectorized.

There are two complications in reducing the number of limbs from five to three:

- While the limbs themselves will be less than 64 bits, their products will not be – some way must be found to perform multiplications producing results of 90 bits or more.
- 130 is not divisible by three, meaning the limbs will not be of equal length.

We will solve the first issue by using a new instruction available on z14. The second issue is not a serious one – we can for instance use limbs of 44, 44, and 42 bits as suggested in the Donna implementation in [4]. However, some caution has to be applied when using the "reduce trick" of multiplying by five. The trick is predicated on the operations being in a fixed radix system. In our case, we would have to act as if the radix was 2^{44}. As the last limb is 42 bits, the remainder has to be shifted by two before it is multiplied by five – or in other words it multiplied by 20 instead of five.

4. A New Approach to Poly1305

The current implementation of the Poly1305 message authenticator uses five 26-bit limbs for the message blocks and r to process the multiplication of two blocks in one iteration. It arranges two sets of five vector registers, V and V' as shown in Figure 1 below. The 26-bit limb size ensures the results of the multiplication do not exceed 64 bits, allowing the computation to be parallelized by using one set of vector registers for two message blocks. However, this multiplication contains several dependencies due to register pressure, does not completely fill up the pipeline and becomes a bottleneck in performance.

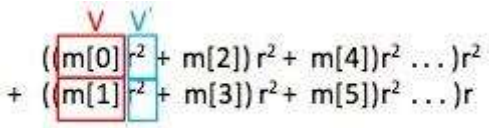


Figure 1

An alternative approach to generate the Poly1305 MAC that Bernstein suggests in his paper involves using r^4 and an additional set of registers to process four message blocks in parallel. He proposes setting up the registers in the conventional, vertical way as shown in Figure 2, with limbs such that the (V*V') and (V''*V''') multiplications can be done in parallel. While this approach promises performance improvement, it is not feasible due to the increased register pressure on the system.

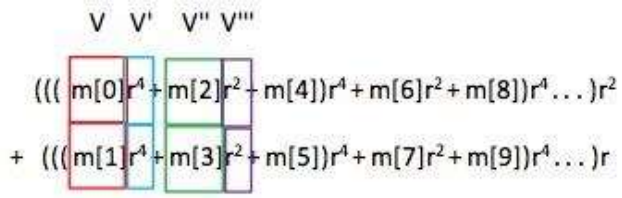


Figure 2

In our implementation, we take a new innovative approach by flipping Bernstein's vertical method to set up the limbs as shown in Figure 3. Instead of five limbs, we use three limbs of 44, 44 and 42-bits. Two sets of three vector registers, V and V' are used to hold the limbs of the upper message blocks, m[0] and m[2] and the lower ones, m[1] and m[3] respectively. The limbs are arranged in a parallel fashion in each half of the vector register. Five more registers, V'' are used for the limbs of r^4 and r^2 that are arranged in a similar fashion as the message blocks. The reason we have five registers for r^4 and r^2 is that for each we precompute two limbs multiplied by 20. Recall that the "reduce trick" requires a multiplication by 20. In practice, this is only required for two limbs, and it is efficient to precompute these.

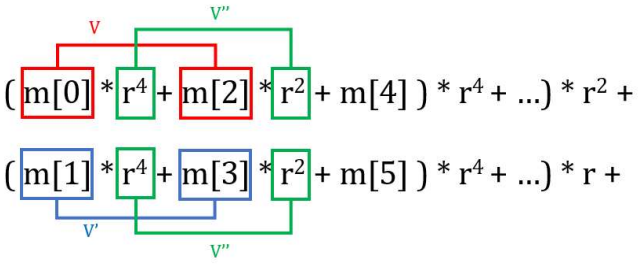


Figure 3

This approach to computing the equation gives us several advantages. Due to the larger limb sizes, it reduces the register pressure by using two sets of three vector registers for four message blocks in place of two sets of five vector registers.

Besides increased limb size, this arrangement allows us to use the Vector Multiply Sum Logical (VMSL) instruction in the z14 architecture to perform two multiplications and two additions in one operation. VMSL is able to achieve this feat by utilizing the floating-point pipeline on the z14. In order to maintain constant time operations (a requirement of cryptographic algorithms) VMSL constrains operands to have fewer than 56 significant bits. With our limb sizes, that constraint is met. Furthermore, the VMSL instruction is significantly faster than the conventional vector multiply that was previously used. By cleverly interleaving computation for the upper and lower limbs and completely filling up the pipeline, we obtain a 3.6 times performance improvement in the multiply portion of the algorithm. In some preliminary profiling of our implementation, we see that the multiply function is no longer a performance bottleneck.

5. Comba Multiplication with VMSL

To perform the 3-limbed multiplication, we set up the equation as shown in Figure 4 using the Comba multiplication described above. Note, in this figure, a notation such as "$m[0]_0 * r^4_0$" refers to the multiplication of the zeroth limb of message block m[0] by the zeroth limb of r^4. The six accumulator registers contain the results for the upper and lower halves of the equation. Each VMSL operation (which performs two multiplies and an addition) is enveloped in a blue box. This leads to an average of 4.5 operations per message block, which is a significant decrease from 12.5 operations per message block that the current implementation uses. By interleaving the multiplications for four blocks, we successfully reduced dependencies and filled up the pipeline. By using larger limb sizes, we also reduced the number of multiplications that were needed per iteration from 25 to 18 and doubled the number of message blocks that were processed per iteration.

The effective scheduling of the VMSL operation is obtained using temporary registers that hold the intermediate results of the Comba multiplication. As shown in Figure 4, all VMSL operations marked with a `T` are issued to temporary registers and the operations marked with an `A` are issued into accumulator registers. This allows us to issue 18 independent VMSLs in sequence completely filling up the pipeline. All temporary results are then added to the accumulator results for the final output. Since VMSL has an 8-cycle latency, no operation waits for an operand to be ready for use and omits internal dependencies in the function. To finish off, the intermediate results are all added into the accumulator registers for the final answer. This scheduling is made possible by freeing up the registers that were previously used to hold the five limbs of the messages and results in a *3.6* × performance improvement in the core loop multiplications of the Poly1305 algorithm.

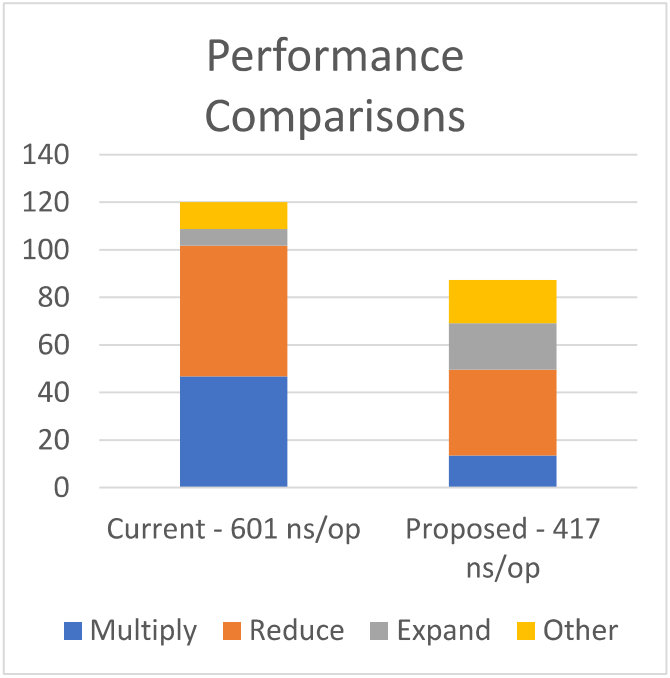
6. Bottleneck analysis

To obtain performance metrics for our implementation of the Poly1305 algorithm on z14 to compare with the current implementation's performance on a z14 machine, we used Go profiling tools to identify and correct specific bottlenecks. Go profiles provide information such as the time taken for a single execution of the program (averaged over multiple runs) and absolute values and percentage of the total program running time spent on specific macros and instructions in the code. These profiles were run on dedicated z14 performance machines for accurate running time information. The Poly1305 program was run 500 million to 1 billion times on a large message of size 2054 bytes and the results were tabulated for the graph shown in Figure 5.

As illustrated in Figure 5, the overall running time of the algorithm has decreased from 601 to 417 ns per Poly1305 operation, showing a *1.4* × improvement. The multiply operation presents a *3.6* × improvement, reduce is improved by *1.6* × but expand deteriorates by about *2.7* ×.

The improvement observed in the main loop multiplications is very encouraging for the performance of Poly1305 and other cryptographic algorithms on z14. However, two important observations have been made on the performance results which show that further work is needed. First, although the main loop multiplications are significantly faster, they are stalled by the expand macro which takes about twice as much time than it took previously. Thus, multiplication, overall, is improved by only about *3.6* ×. Second, a new bottleneck has emerged: reduction operations. When we examine the proportion of CPU cycles that were spent in the various portions of the algorithm, both before and after our optimization (on z14), the magnified impact of reduction can clearly be seen.

Figure 5



$$P_0[0] = \overbrace{(m[0]_0*r^4_0 + m[2]_0*r^2_0)}^{A} + \overbrace{(m[0]_1*r^4_2 20 + m[2]_1*r^2_2 20)}^{T} + \overbrace{(m[0]_2*r^4_1 20 + m[2]_2*r^2_1 20)}^{T}$$

$$P_0[1] = \overbrace{(m[0]_0*r^4_1 + m[2]_0*r^2_1)}^{A} + \overbrace{(m[0]_1*r^4_0 + m[2]_1*r^2_0)}^{T} + \overbrace{(m[0]_2*r^4_2 20 + m[2]_2*r^2_2 20)}^{T}$$

$$P_0[2] = \overbrace{(m[0]_0*r^4_2 + m[2]_0*r^2_2)}^{A} + \overbrace{(m[0]_1*r^4_1 + m[2]_1*r^2_1)}^{T} + \overbrace{(m[0]_2*r^4_0 + m[2]_2*r^2_0)}^{A}$$

$$P_1[0] = \overbrace{(m[1]_0*r^4_0 + m[3]_0*r^2_0)}^{A} + \overbrace{(m[1]_1*r^4_2 20 + m[3]_1*r^2_2 20)}^{T} + \overbrace{(m[1]_2*r^4_1 20 + m[3]_2*r^2_1 20)}^{T}$$

$$P_1[1] = \overbrace{(m[1]_0*r^4_1 + m[3]_0*r^2_1)}^{A} + \overbrace{(m[1]_1*r^4_0 + m[3]_1*r^2_0)}^{T} + \overbrace{(m[1]_2*r^4_2 20 + m[3]_2*r^2_2 20)}^{T}$$

$$P_1[2] = \overbrace{(m[1]_0*r^4_2 + m[3]_0*r^2_2)}^{A} + \overbrace{(m[1]_1*r^4_1 + m[3]_1*r^2_1)}^{T} + \overbrace{(m[1]_2*r^4_0 + m[3]_2*r^2_0)}^{A}$$

Figure 4

To improve the performance of reduce, and possibly other operations as well, the approach we are taking is to use *parallel reduce* as described in Appendix 2 which allows us to pipeline our instructions better and gives us more possibilities for optimal scheduling. It is our observation that vector code, with its heavy reliance on pipelining for good performance, is particularly sensitive to scheduling. Producing an optimal or near optimal schedule can result in dramatic improvements in performance. The approach we take to generating optimal schedules is to apply the Coconut Domain Specific Language (DSL) as outlined below. In previous applications, Coconut has been able to produce dramatic performance improvements. For example, in [5], coconut produced an average 4.5 increase in throughput for implementations generated and scheduled by Coconut over optimized C implementations. However, Coconut has not yet been used on integer vector code, so work needs to be done to apply it to this new domain.

7. Rapid Prototyping in Coconut

The development environment known as Coconut (an abbreviation for COde CONstructing User Tool), contains an extensible domain-specific language (DSL) embedded in Haskell. The DSL supports interactive prototyping and unit testing, simplifying the process of designing efficient implementations of common patterns [6]. An overview of the prototyping process as presented for this paper is illustrated in Figure 6.

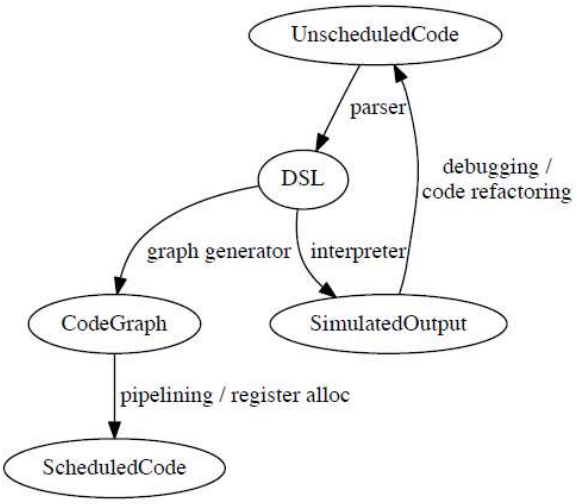


Figure 6

8. Parsing & DSL Generation for Z Assembly Programming in GO

To assist in development, a parser for macros containing Z assembly in GoLang's format (slightly different than standard) was developed. Macros containing Z assembly code are parsed into a purely functional Haskell embedded DSL reflecting SSA (Single Static Assignment) form, as illustrated by the following example code:

```
#define INLINED_CODE(a0,b0,c0,..) \
        vadd a0,b0,a0 \
        vsub a0,c0,a0 \
        ...

==> Parser / DSL Generation ==>

inlined_code (a0,b0,c0,...) = let
        a0_0 = vadd b0 a0
        a0_1 = vsub c0 a0_0
        ...
        in [a0_0,a0_1,...]
```

The generated Haskell function takes a tuple of inputs corresponding to the inputs of a Go macro, such as *Reduce*, and returns a list of intermediate results, which can then either be simulated to extrapolate results for debugging purposes or used to generate a corresponding code graph. Example simulated output can be easily formatted like so:

(0xFFFFFFFF, vadd b0 a0) (0x82AFFFFF, vsub c0 a0_0) ...

9. A DSL for Prototyping & Optimized Assembly Code Generation

The Haskell Embedded DSL provides an interface for experts in high performance numerical software to provide processor-specific (in this case Z) implementations of mathematical functions, with the following goals [6]:

1. The language supports safety and is familiar to mathematicians.
2. Common code construction tasks are simplified.
3. The user is insulated from details of the target intermediate language not relevant to their task.
4. The environment supports exploration and rapid prototyping.
5. Alternatives for instruction selection can be provided by the user.

Of particular interest to this project, is the ability to interpret and debug intermediate code without consideration of register allocation or pipelining constraints. By representing macros as pure functions embedded in Haskell, one can easily manipulate and compose these functions and perform unit testing from a high-level setting (i.e. within Haskell).

After debugging / refactoring of code using the built-in interpreter Coconut provides, the same code used for interpretation (i.e. the generated pure function) can be used to generate a corresponding code graph. Coconut then performs optimizations over the code graph, including an approximation algorithm based approach to instruction scheduling as described in [7], to generate near-optimal scheduled assembly code. Our next goal is to apply these techniques to the portions of Poly1305, such as *Reduce* which have proven to be performance bottlenecks.

10. Instruction Scheduling

We have made preliminary investigations with a scheduler based on continuous optimization, which was developed as part of a project to create random algorithms for instruction scheduling. The project grew out of a desire to make better use of out-of-order

execution resources on hardware like z14. Specifically, for complicated basic blocks, the number of registers available in the instruction set architecture is likely to be much smaller than the number of physical registers, and also, to be insufficient to implement the basic block without spilling values onto the stack.

A schedule which, taking latency into account, is executable in order without stalls, does not use more than the architected number of registers. So to avoid spilling, it is better to schedule instructions "too early" to force the out-of-order hardware to make better use of the physical registers. As a simple heuristic, an instruction which consumes more values than it produces should be scheduled as soon as possible, because it will reduce logical register pressure. One approach to doing this is to iteratively apply min-cut, to cut the dependency graph into strongly connected components with few inter-component connections. Each strongly connected component can be scheduled together, and then the components can be interleaved. This is the idea behind Kriston Costa's MSc thesis [7]. Unfortunately, there are other limited resources in the Central Processing Unit besides the register file. Balancing scheduling heuristics is a known hard problem, of continuing research interest [10,11].

In contrast, continuous optimizations can incorporate an arbitrary number of constraints and goals to be balanced, in the form of penalty functions [12]. This suggests that we formulate a continuous optimization problem to guide us as to the right balance between competing goals, and then use that balance to guide a random algorithm.

As a first experiment, we have formulated a number of constraints and penalty functions, generated optimization problems from the dependency graph of a basic block, and simply rounded continuous values to their nearest integer to create a solution to the underlying discrete problem. The dependency graph itself gives us hard constraints on the relative dispatch time of instructions. To avoid running out of various processor resources, the same constraints can be made more strict. We have experimented with many types of constraints, including non-convex constraints, which introduce the possibility of finding multiple locally-optimal solutions.

An example of a simple soft constraint, is the I/O penalty. For each instruction, we add a penalty to the objective function which is a sum of the dispatch times of its inputs inversely weighted by the number of other consumers minus the sum of the dispatch times of its consumers inversely weighted by the number of values each consumer consumes.

The time consumed by such optimizations will necessarily be longer than for conventional compiler optimizations, but for basic blocks which execute frequently, this may be worthwhile. It is certainly worthwhile for performance-critical basic blocks which are distributed with the compiler or operating system, and which are frequently executed by almost all users of a particular architecture. In practice, we have found that the time for solving basic blocks with a few hundred instructions is well under a minute in all cases.

For the Poly1305 implementations under investigation, we have produced a 7% reduction in processing time, but we expect that we can do much better.

11. Conclusion

In this study, we have re-structured the computation for polynomial evaluation in a prime field required for Poly1305 message authentication in order to accelerate its computation on the z14. In doing so we have successfully vectorized multiplication of large limbs (44 and 42 bits) in a manner which best exploits pipelined multiplication via the floating-point multiplication units of z14, using the newly designed integer instruction VMSL. In doing so we have exposed other bottlenecks in the algorithm, which will subsequently attempt to solve with optimal instruction schedules as devised by the Coconut DSL system. In the future, our work on Poly1305 crypto on z will be broadened to other cryptographic algorithms. For example, in elliptic curve cryptography, application of Karatsuba-Ofman methods to the P224 curve would result in the multiplication of nine 56 bit numbers [8]. That would appear to be an opportunity to apply VMSL, as it can do 56 bit multiplications in constant time.

12. Acknowledgements

We would like to thank Robert Enenkel who first explored the Coconut DSL on IBM Z, Michael Munday who wrote the Z13 implementation of the Poly1305/ChaCha20 encryption cypher and Jonathan Bradbury for designing the VMSL instruction and helping us understand it.

13. References

1. Bernstein, Daniel J. “The Poly1305-AES message authentication code”, in Proceedings of the 12th international conference on Fast Software Encryption, March 29, 2005.
2. Vaillant, Loup. “The design of Poly1305”, Issued January, 2017. http://loup-vaillant.fr/tutorials/poly1305-design.
3. *Principles of Operation, z14*, IBM (A22-7832-10).
4. Moon, Andrew. “floodyberry/poly1305-donna”, Issued March 28, 2016. https://github.com/floodyberry/poly1305-donna.
5. Anand, Christopher Kumar and Kahl, Wolfram. “An optimized cell BE special function library generated by Coconut. IEEE Transactions on Computers, 58(8):1126–1138”, Issued 2009.
6. Anand, Christopher Kumar and Kahl, Wolfram. “A domain-specific language for the generation of optimized SIMD-parallel assembly code SQRL Report, 43”, Issued 2007.
7. Costa, Kriston. MSc. thesis “An Approximation Algorithm Based Approach to Instruction Scheduling”, McMaster University, 2016.
8. Hankerson, D., Menezes, A. J., and Vanstone, S. “Guide to Elliptic Curve Cryptography”, Springer, 2004.
9. Comba, P., Exponentiation cryptosystems on the IBM PC. IBM Systems Journal, 29:526–538, 1990.
10. Valluri, M. G. and Govindarajan, R. Evaluating Register Allocation and Instruction Scheduling Techniques in Out-Of-Order Issue Processors. In Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99). IEEE Computer Society
11. Nobre, R., Martins, L. G. A., and Cardoso J. M. P. 2016. A graph-based iterative compiler pass selection and phase ordering approach. SIGPLAN Not. 51, 5 (June 2016), 21-30.
12. J. Nocedal, and Wright S. J., Numerical Optimization, Springer New York, 2006
13. Bernstein, Daniel J. and Schwabe, Peter. “NEON crypto”, in CHES'12 Proceedings of the 14th international conference on Cryptographic Hardware and Embedded Systems, September 09, 2012.

Appendix 1: VMSL on z14

The Vector Multiply Sum Logical (VMSL) instruction is a part of the z14 architecture [3]. It has an 8-cycle latency, and can be efficiently pipelined to obtain performance boosts that are important to our work. The instruction takes four vector registers as input and an element size control mask. The mask can be omitted by using the extended mnemonic, VMSLG that specifies the element size as grand (64 bits). Figure 7 describes how a VMSLG instruction works.

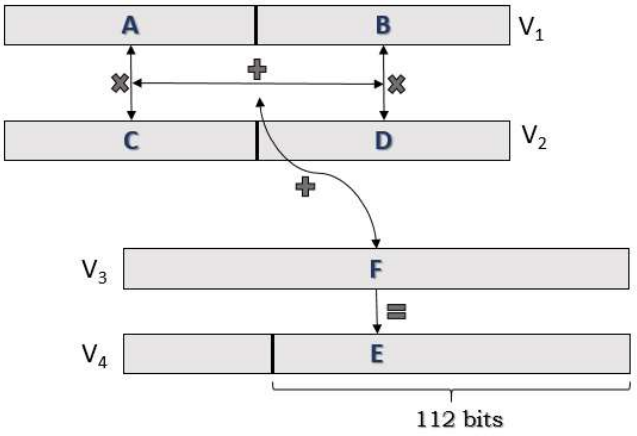


Figure 7

The 64-bit upper and lower halves of the vector register, V_1 are multiplied with the corresponding halves of V_2. The two intermediate products are then added together and further added to V_3 and the result is placed into V_4. VMSL uses floating points vector units for its computation which increases the possible use cases of the instruction, but restricts the number of significant bits in the operands to less than or equal to 56 for constant time. Therefore, for crypto purposes we must ensure that the operand sizes are always less than 56 bits to run in constant time and avoid timing attacks on our code.

Appendix 2: Parallel Reduce

As mentioned in Section 2, dependencies can be mitigated in reduction by reducing in alternating limbs. For example, in the Neon five-limb approach [13], reduction can proceed as thus:

$h_0 \rightarrow h_1$, $h_3 \rightarrow h_4$, $h_1 \rightarrow h_2$, $h_4 \rightarrow h_0$, $h_2 \rightarrow h_3$, $h_0 \rightarrow h_1$, $h_3 \rightarrow h_4$. However, in our case there are only three limbs for each accumulator, which makes this non-adjacent approach difficult. Our first attempt with three limbs was simple serial reduction:

$$h_0 \rightarrow h_1, , \; h_1 \rightarrow h_2, \; h_2 \rightarrow h_0, \; h_0 \rightarrow h_1,$$

However, this version of reduce was a very significant bottleneck, far eclipsing multiplication as a bottleneck. In order to, mitigate this issue, we developed a new form of reduce: parallel reduce. In parallel reduce, the reduction steps are carried out on each limb independently of the other limbs. First carry values are computed (without intervening additions to the subsequent limb). For three limbs we will have three carry values. Then the carries are added (and multiplied by 20 in the case of the $h_2 \rightarrow h_0$ transformation), again independently. The whole reduction process is then repeated. This virtually eliminates any dependency delays that hamper the reduce operation.

It is worth noting that it can be established that this parallel reduction can be shown to be effective. It is expected that after parallel reduce, there will be at most one bit of overflow in any limb. We establish this through the following process. Assume the worst as an initial state: that all three limbs overflow by 1 bit, and that all new message blocks are fully-filled out with ones. Perform the multiplication/addition steps of the algorithm, then a parallel reduce. We find that in this case the number of limbs that overflow by a single bit is two (the third doesn't overflow). When we run it through another iteration of the algorithm (again assuming the worst possible message blocks), we find that the result has no overflowing limbs.

Conversely, if we start with no limbs overflowing, but filled out with the maximal ones, and assuming that all incoming message blocks are always filled out maximally, we find that one limb overflows by one bit, and two do not overflow. This situation persists through one more iteration of the algorithm, but in the subsequent iteration, there are no overflows. Thus, parallel reduce provides for a monotonic decrease in the number of overflowing limbs as the algorithm proceeds. We have never observed an overflow in excess of one bit. This means that in practice, our limbs are never greater than 45 bits, which is well short of the 56 bits required for constant-time multiplication.